LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-641294

Optimal Hierarchical Layouts for Cache-Oblivious Search Trees

P. Lindstrom, D. Rajan

July 22, 2013

IEEE International Conference on Data Engineering
Chicago, IL, United States
March 31, 2014 through April 4, 2014

Disclaimer

Optimal Hierarchical Layouts for Cache-Oblivious Search Trees

Peter Lindstrom and Deepak Rajan
Center for Applied and Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California 94550, USA
{pl,rajan3}@llnl.gov

***Abstract*—This paper proposes a general framework for generating cache-oblivious layouts for binary search trees. A cache-oblivious layout attempts to minimize cache misses on any hierarchical memory, independent of the number of memory levels and attributes at each level such as cache size, line size, and replacement policy. Recursively partitioning a tree into contiguous subtrees and prescribing an ordering amongst the subtrees, *Hierarchical Layouts* generalize many commonly used layouts for trees such as in-order, pre-order and breadth-first. They also generalize the various flavors of the van Emde Boas layout, which have previously been used as cache-oblivious layouts. Hierarchical Layouts thus unify previous attempts at deriving layouts for search trees.**

The paper then derives a new locality measure (the Weighted Edge Product) that mimics the probability of cache misses at multiple levels, and shows that layouts that reduce this measure perform better. We analyze the various degrees of freedom in the construction of Hierarchical Layouts, and investigate the relative effect of each of these decisions in the construction of cache-oblivious layouts. Optimizing the Weighted Edge Product for complete binary search trees, we introduce the MINWEP layout, and show that it outperforms previously used cache-oblivious layouts by almost 20%.

I. INTRODUCTION

In today's computer architectures, the memory hierarchy is becoming increasingly complex, both in terms of number of levels and difference in performance from one level to the next. As a result, algorithms and data structures that are designed for flat (or even two-level) memory with uniform access times can result in significantly suboptimal performance. In this paper, we are interested in improving memory access locality for search trees via data reordering. The classic search tree is the B-tree [1], which has been designed for a two-level cache hierarchy, and is usually optimized for a particular block transfer size (*e.g.*, a cache line or disk block). B-trees have been extended to more than two levels of cache hierarchy [2], but it is not clear if they can be successfully optimized for an arbitrarily complex multi-level cache hierarchy, with one level per transfer block size. Furthermore, B-trees are known to perform poorly when the nodes of the search trees are of different sizes (*e.g.*, when the search keys are variable-length) [3]. As a result, cache-oblivious search trees have been suggested in the literature. In this paper, we present a new locality measure that can be used to derive cache-oblivious data structures. Focusing our attention on search trees, we show how optimizing our locality measure results in better cache-oblivious search tree layouts than prior layouts.

The fundamental structure commonly employed for cache-oblivious search trees is the van Emde Boas layout. First introduced by Prokop [4], these recursively defined layouts are similar to van Emde Boas trees, hence the name. In [5], this layout was shown to result in much better binary search times than simpler orderings such as breadth-first and depth-first pre- and in-order. Repeating these experiments to measure the L1 cache miss rates for these orderings as well as our proposed layout, MINWEP, we present the results as a function of tree height in Figure 1. Our experiments confirm that the van Emde Boas trees perform significantly better than the simpler orderings. As observed in [5], the in-order layout performs particularly poorly, with a cache miss rate close to 100%. This can be attributed to a limited associativity of the cache; for complete binary trees, the in-order layout arranges the nodes at the top of the tree at positions that are large powers of two apart from each other. We also observe that our proposed layout MINWEP consistently reduces L1 cache misses by about 20% compared to the van Emde Boas layout.

Minor variants of these van Emde Boas layouts have since been used in a variety of other settings. In [6], the authors introduce a very similar layout that differs only in how the tree is partitioned at each branch of the recursion. Using this layout as the basic building block, they present dynamic search trees, and refer to these as cache-oblivious B-trees. In [7], the authors provide bounds on the asymptotic cost of cache-oblivious searching. By analyzing a generalized version of

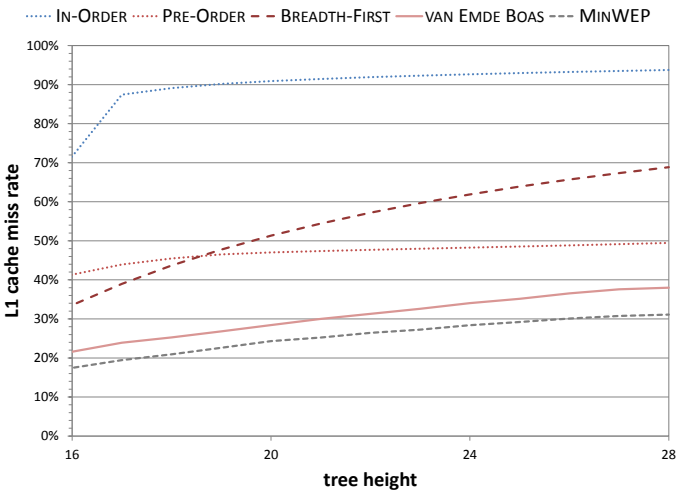


Fig. 1: L1 cache miss rate as a function of tree height for binary searches.

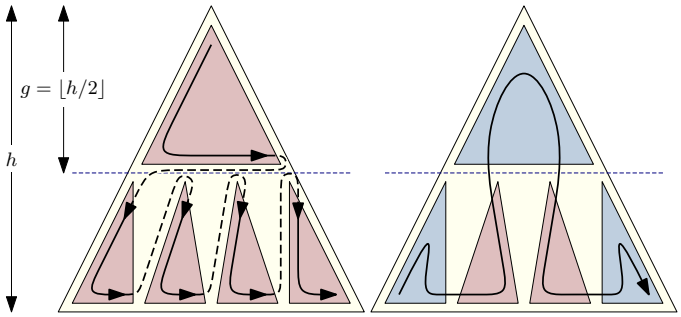


Fig. 2: Hierarchical layouts of a binary tree formed by recursive partitioning into contiguous subtrees. In-order (blue) and pre/post-order (red) subtrees are shown for the van Emde Boas layout (left) and our MINWEP layout (right). The arrows indicate the ordering within and among the subtrees.

the van Emde Boas layout, they provide a modified version that is arbitrarily close to the asymptotic bound. In [3], [8], the authors address the problem of building cache-oblivious layouts of search trees with variable-sized search keys. They use a modified version of the van Emde Boas layout in which the tree is partitioned differently. In [9], the authors present two cache-oblivious streaming B-trees – data structures that implement cache-oblivious search trees optimized for dynamic insertions and deletions. Again, these rely on a version of the van Emde Boas layout with a slightly different partitioning scheme. We note that cache-oblivious data structures are not limited to search trees. They have been proposed in a variety of settings, some of which include hash tables [10], meshes [11], [12], and Bloom filters [13].

A. Contribution: Cache-oblivious Hierarchical Layouts

We describe a general framework for generating search tree layouts, and present new orderings from this framework that result in better cache-oblivious search tree layouts than those suggested in the literature. We refer to all layouts that fit the new framework as *Hierarchical Layouts*.

Consider a tree T of height h, *i.e.*, with h levels of nodes. Counting the levels in a tree of height h from top to bottom, the root is on level 0 and the leaves are on level $h-1$. Observe that level i has 2^i nodes. For ease of exposition, we restrict our discussion to complete binary trees; therefore the number of nodes is $2^h - 1$.

Any Hierarchical Layout can be described recursively as follows: Partition T by cutting it horizontally between level $g-1$ and g, which results in a top subtree of height g with 2^{g-1} leaves and 2^g bottom subtrees (2 for every leaf in the top subtree) of height $h-g$. Any relative ordering of the recursive subtrees that arranges them contiguously in memory constitutes a Hierarchical Layout. Two obvious relative orderings (out of the exponentially many choices) are those where the top subtree is arranged either in the middle of all the bottom subtrees (in-order), or at one end (pre-order). Figure 2 illustrates these two cases using the colors blue and red, for in- and pre-order, respectively. The node ordering within each subtree is given by recursive application of this decomposition, until each subtree consists of a single node.

The effectiveness of any particular Hierarchical Layout as a cache-oblivious search tree depends on the relative ordering of subtrees and on the height of the top subtree g. In this paper, we focus on finding optimal cuts and orderings of the subtrees to maximize locality and minimize cache misses, and propose a new cache-oblivious Hierarchical Layout.

We also show that the widely used van Emde Boas layouts are a special case of Hierarchical Layouts; therefore any cache-oblivious search tree data structure that utilizes a van Emde Boas layout can be improved by switching to our proposed layout. In particular, the performance of various cache-oblivious dynamic search trees (cache oblivious B-trees that utilize variants of van Emde Boas layouts as building blocks [3], [6], [8], [9]) can be improved by utilizing our proposed Hierarchical Layout (MINWEP) as the basic building block instead. The main take-home message of this paper is that the widely used version of the van Emde Boas layout is **not** the best Hierarchical Layout. Significantly better cache-oblivious layouts can be obtained by considering Hierarchical Layouts that minimize our locality measure.

Section II presents a nomenclature that allows us to concisely characterize Hierarchical Layouts. In Section III, we motivate some simple improvements to the van Emde Boas layout. Section IV describes a mathematical measure of locality for tree orderings that correlates well with cache miss ratios, thus resulting in cache-oblivious layouts. We analyze which Hierarchical Layouts perform better with respect to this new measure, the *Weighted Edge Product*. We improve the layout further in Section V by deriving MINWEP, the Hierarchical Layout that minimizes the Weighted Edge Product. Figure 2 presents a simplified illustration of the key differences between the van Emde Boas layout and MINWEP. Our experiments indicate that MINWEP consistently improves performance by about 20% compared to the layouts described in the literature.

II. HIERARCHICAL LAYOUTS: NOTATION AND NOMENCLATURE

At any branch of the recursion describing a Hierarchical Layout, we use A to denote the top subtree and L_A to denote the set of leaves in the top subtree A. Given a leaf node x in L_A, we say that a bottom subtree formed by a child c of x and the descendants of c is a *child subtree* of x.

A Hierarchical Layout is given entirely by (1) the height at which the tree is partitioned, (2) the position of the top subtree relative to the bottom subtrees, and (3) the relative ordering of the bottom subtrees. This definition allows for a very large combination of cut heights and orderings. For this reason, we impose additional restrictions; the motivation behind some of them will become clear later in the paper. We refer to layouts belonging to this restricted set as Recursive Layouts because they can be categorized entirely using a small set of recursive rules and parameters, allowing for a more compact nomenclature than the more general Hierarchical Layout.

In a Recursive Layout, at any branch of the recursion that cuts a subtree into its top subtree A and the corresponding bottom subtrees, we enforce the following restrictions. (a) The top subtree A is arranged either in-order or pre-order. (b) The top subtree obtained in the partitioning of A must be arranged relative to the bottom subtrees in the same fashion as A. (c) If A is arranged in-order, we choose the children of the leftmost 2^{g-2} leaves in L_A to be the bottom subtrees on the left of the

top subtree. (d) If any bottom subtree is arranged in-order, all bottom subtrees that are arranged further away from A are also arranged in-order. (e) Looking outwards from A, the bottom subtrees are either ordered in the same order as that of the parent leaves L_A or in the reverse order. (f) If A is arranged pre-order, then it is placed on the side of the bottom subtrees that is closer to its parent leaf. Thus, we use pre-order layouts to refer to both pre-order and post-order arrangements of the top subtree, depending upon the context. (g) The cut height g is a function only of the height of the subtree and whether the subtree is arranged in- or pre-order.

Based on the preceding discussion, we present a new nomenclature for categorizing Recursive Layouts, an important subset of Hierarchical Layouts. A Recursive Layout is categorized as $\mathcal{P}$ for pre-order and $\mathcal{I}$ for in-order to indicate the arrangement for the outermost branch of the recursion (when we cut the tree T itself). At each branch of the recursion, the position of the first in-order bottom subtree, counting outwards from the top subtree, is indicated as a subscript. If all the bottom subtrees are arranged pre-order, then we denote this by ∞. The cut height g (as a function of the height of the subtree h) is indicated as a superscript. We indicate a layout where the bottom subtrees are arranged in reverse order of the leaves L_A using the $\sim$ symbol on top.

Bringing this all together, we see that $\widetilde{\mathcal{I}}_2^{\lfloor h/2 \rfloor}$ is the Recursive Layout that always cuts a subtree of height h at height $\lfloor h/2 \rfloor$, arranges the top subtree in the outermost branch of the recursion in-order, arranges all bottom subtrees in the reverse order of the top subtree leaves, and arranges the first bottom subtree pre-order and all the other bottom subtrees in-order. In Table I, we categorize all the layouts we consider in this paper using this nomenclature. All the layouts described in this paper belong to the restricted set of Recursive Layouts.

III. Cache-oblivious Hierarchical Layouts

In this section, we motivate better cache-oblivious orderings within the framework of Hierarchical Layouts. First, we review the van Emde Boas layouts used in the literature, which are a special case of Hierarchical Layouts. In Prokop's ordering [4], the subtrees are cut at height $g = \lfloor h/2 \rfloor$, the top subtree is placed before the bottom subtrees, and then this ordering strategy is applied recursively to each subtree (see Figure 2). The bottom subtrees are arranged in the same order as their parent leaves L_A, from left to right. Henceforth, we refer to this version of the van Emde Boas layout as the pre-order van Emde Boas layout, and denote it as PRE-vEB. In our nomenclature, PRE-vEB is $\mathcal{P}_\infty^{\lfloor h/2 \rfloor}$ (see Table I).

Figure 7f illustrates PRE-vEB for a tree of height 6. The number inside each node is its position in the layout, ranging from 1 to 63. Observe that at every branch of the recursion, the top subtree is arranged pre-order. In the outermost branch of the recursion, the nodes in the top three levels are arranged first (positions 1 to 7). Figure 7 also indicates the length of each edge, *i.e.*, the difference in position of its nodes, using lines whose thickness is inversely proportional to the length.

In Bender's layout [6], the authors set $g = h - 2^{\lceil \log_2(h/2) \rceil}$. In other words, the height of the bottom subtrees equals the largest power of two smaller than h. The authors refer to their layout as a van Emde Boas layout since it is similar to the one introduced in [4]. Nevertheless, we make the distinction that only Hierarchical Layouts with $g = \lfloor h/2 \rfloor$ are van Emde Boas layouts. BENDER is identical to PRE-vEB for trees whose height is a power of two. For all other heights, BENDER layouts have smaller top subtrees, compared to PRE-vEB. Figure 7l illustrates Bender's layout. Observe that the nodes in the top 2 levels are arranged next to each other, indicating a cut height of 2 at the outermost branch of the recursion. In our nomenclature, BENDER is $\mathcal{P}_\infty^{h-2^{\lceil \log_2(h/2) \rceil}}$ (see Table I).

We will see later that Hierarchical Layouts also include all the simple and commonly used layouts such as in-order, pre-order, and breadth-first. One can think of cut heights $g = 1$ and $g = h - 1$ as the extreme cases, corresponding to these simple layouts. We will also show that cache-oblivious layouts are obtained by cutting the tree near the center, with g approximately equal to $h/2$.

A. Evaluating layouts using block transitions

To compare Hierarchical Layouts, we will estimate the number of cache misses for a particular cache block size and layout as follows. Consider a cache consisting of a single block that can hold N data elements, and which is backed by a larger memory consisting of several such blocks. (In practice caches tend to hold more than one block, but that would unnecessarily complicate our derivation.) Let i and j be data elements stored in blocks $B(i)$ and $B(j)$, respectively, and let ℓ_{ij} denote the difference in position of i and j on linear storage. For ease of exposition, we set $\ell_{ij} = \ell$. Suppose i is accessed first, bringing $B(i)$ into the cache. We wish to estimate the probability of a cache miss when j is accessed next. Clearly, if $\ell \geq N$, then a cache miss is inevitable, since then i and j are stored in different blocks. When $\ell < N$, the likelihood of a cache miss depends on the positions of i and j within their blocks. In absence of further information, we will assume that the position of i within $B(i)$ is distributed uniformly, and similarly for j. (Even in practice, modern operating systems allocate memory blocks with nearly arbitrary alignment.) Hence, there are ℓ out of N possible alignments that separate i and j into different blocks, and the probability of a cache miss occurring when j is accessed is therefore

$$M_N(\ell) = \begin{cases} \frac{\ell}{N} & \text{if } \ell \leq N \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

To represent a particular access pattern on the data, we use the notion of an *affinity graph*, as in [11], [14]. We model the data elements as nodes V in a graph $G(V, E)$, with an undirected edge indicating a nonzero likelihood that its two nodes be accessed in succession. The affinity between i and j may be expressed in terms of a weight $w_{ij} = w_{ji} > 0$. Let A denote the matrix of affinities, such that $a_{ij} = w_{ij}$ if $ij \in E$ and $a_{ij} = 0$ otherwise. We model data accesses as a Markov chain random walk on G with transition matrix $P = D^{-1}A$, where D is the diagonal matrix with $d_{ii} = \sum_j a_{ij}$. If G is strongly connected, as is the case for binary trees, then it is well-known that the probability $\Pr(X_t = i, X_{t+1} = j)$ of being in state i and transitioning to state j equals $\frac{w_{ij}}{W}$, where $W = \sum_{ij \in E} w_{ij}$. In other words, the probability of accessing two data elements in succession is proportional to the weight of the edge connecting them.

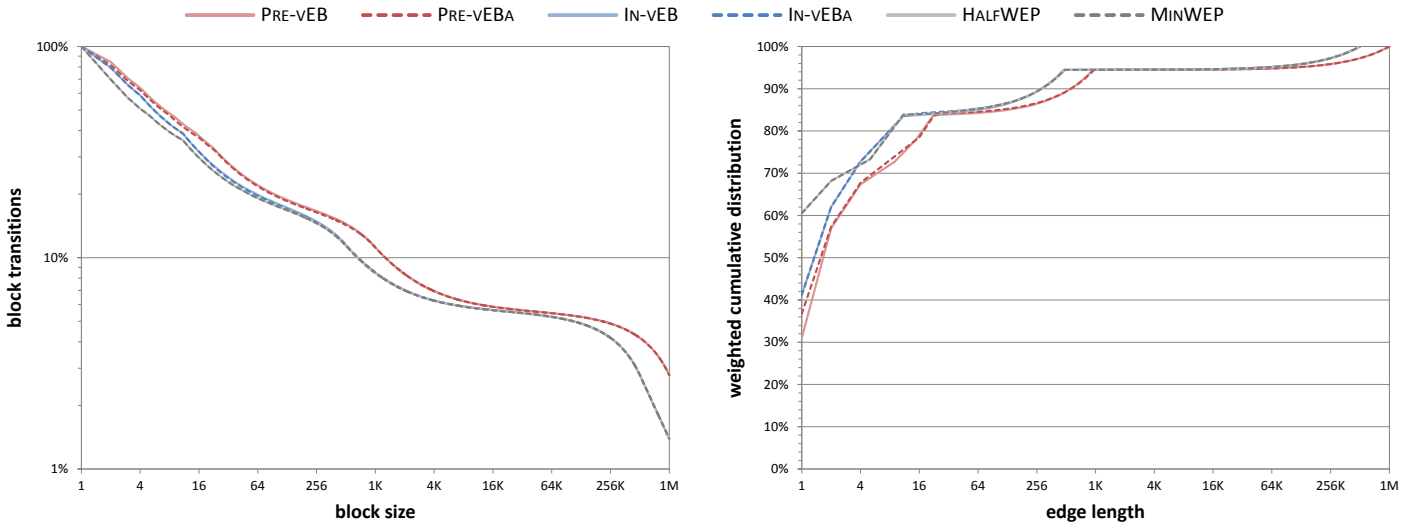


Fig. 3: Two locality measures for several layouts of a tree of height $h = 20$. Left: Block transitions β as a function block size (lower is better). Right: Cumulative distribution of edge weights as a function of edge length (higher is better).

In a binary search tree T, the affinity graph is the search tree itself, and the search for a particular element results in a walk from the root on level $d = 0$ of the tree to the node representing the element. Therefore, only the node searched for and its ancestors are visited, beginning with the root, and thus nodes near the top of the tree are more likely to be visited than nodes near the bottom. Assuming each node is equally likely to be searched for, the likelihood of traversing a given edge between levels $d-1$ and d in a tree of height h is

$$p_{d,h} = \frac{V_{h-d}}{V_h} = \frac{2^{h-d}-1}{2^h-1}, \tag{2}$$

where V_h is the number of nodes in a complete binary tree of height h. For an edge ij between levels $d-1$ and d, we set $w_{ij} = p_{d,h}$, which ensures that the probability of accessing two data elements in succession is proportional to the weight of the edge connecting them.

Given this probability of accessing any two nodes in succession, the expected fraction of consecutive accesses that will result in a cache miss for a particular block size N is

$$\beta(N) = \frac{1}{W} \sum_{ij \in E} w_{ij} M_N(\ell_{ij}) \tag{3}$$

For any given layout and block size, we refer to β as the *Percentage of Block Transitions*. If one layout dominates another for all block sizes under this metric, then clearly it will result in a better cache-oblivious layout.

B. In-order Hierarchical Layouts

Consider the in-order van Emde Boas layout, denoted as IN-VEB, that arranges all bottom subtrees in-order and in the same relative order as that of their parent leaves L_A. In our nomenclature, IN-VEB is $\mathcal{I}_1^{\lfloor h/2 \rfloor}$ (see Table I). Figure 7e illustrates IN-VEB for a tree of height 6. Observe that at each branch of the recursion, the top subtree is arranged in-order. For instance, the nodes on the top three levels are ordered in the middle of the layout, from positions 29 to 35. To compare IN-VEB with the pre-order van Emde Boas layout that arranges all subtrees pre-order (PRE-VEB), we consider the percentage of block transitions β.

Figure 3(left) plots β for PRE-VEB and IN-VEB as a function of block size for a tree of height 20. We see that IN-VEB dominates PRE-VEB for every block size. Interestingly, at very large block sizes, IN-VEB is much better than PRE-VEB. We have observed the same dominance for trees of other heights. In fact, for large block sizes, IN-VEB compares well with MINWEP, which we introduce later as the optimal cache-oblivious Recursive Layout for binary search trees. Looking at the weighted cumulative distribution, which measures the total weight of all edges up to a certain length, we see the same dominance. Again, we observe that IN-VEB is indistinguishable from MINWEP for large edge lengths.

Figure 4(bottom left) plots β for IN-VEB and PRE-VEB as a function of tree height for a block size of 2, 5, and 16 nodes. With 4-byte nodes, a block size of 16 nodes mimics a cache line size of 64 bytes. We see that IN-VEB dominates PRE-VEB for all tree heights, but is dominated by MINWEP. In our experiments, we observed similar results for other block sizes. We observe the same relative dominance (between MINWEP, IN-VEB, and PRE-VEB) in L1 and L2 cache miss rates in Figure 4(bottom right). Interestingly, MINWEP results in even fewer L1 cache misses than the number of L2 cache misses for PRE-VEB, suggesting that MINWEP is a significantly better layout than PRE-VEB, the suggested layout in the literature.

The true measure of any of these layouts is the average time taken to find any node in the search tree (see Section V-F for more details on the experimental setup). To ensure that the wall clock search time is not affected by the time taken to compute the position of a node in the layout, we store two child "pointers" with each node. For this reason, we also refer to the search time as explicit, or pointer-based search time. Illustrated in Figure 4(top right), we see the same behavior as before. IN-VEB is significantly better than PRE-VEB, but is marginally worse than MINWEP. On average, MINWEP is about 5% better than IN-VEB and almost 20% better than PRE-VEB. The sudden uptick at $h = 32$ is due to NUMA

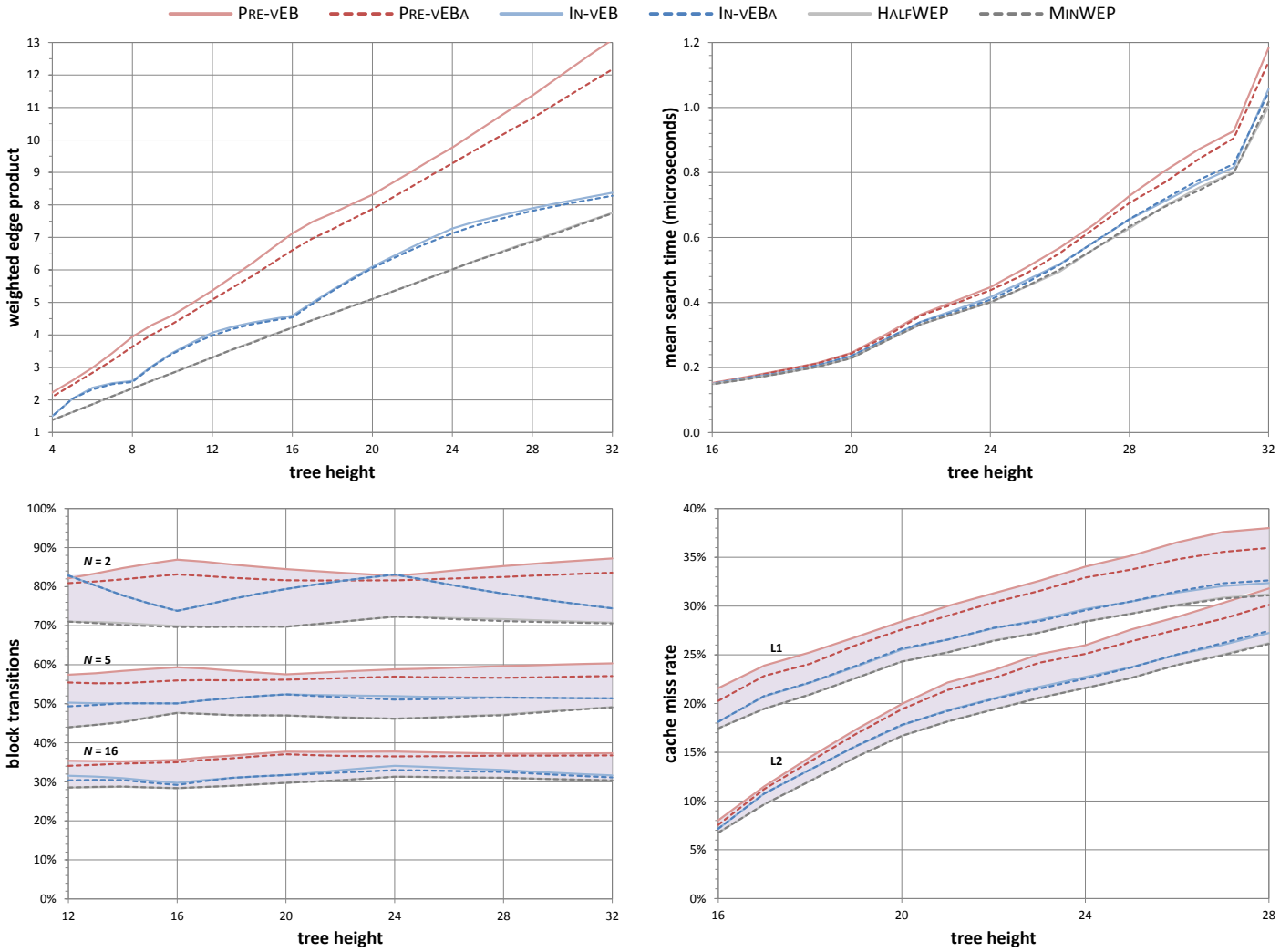


Fig. 4: Performance metrics as a function of tree height for several hierarchical layouts. Top left: Weighted edge length product ν_0. Top right: Wall clock search time. Bottom left: Block transitions for blocks of $N \in \{2, 5, 16\}$ nodes. Bottom right: L1 and L2 cache miss rate.

misses. Our experiments were run on a machine with two memory banks of 48 GB each, and we need 64 GB of RAM to store a tree of height $h = 32$, generating a lot of traffic across the NUMA memory banks. The plots in Figure 4(bottom) therefore indicate that the percentage of block transitions (β) correlates very well with cache miss ratios, and is therefore a good indicator of the quality of a layout. In Section IV, we mathematically derive a new locality measure, the Weighted Edge Product ν_0, which is independent of the block size N and correlates even better with these measures and performance metrics. In Figure 4(top left), we see that IN-VEB has much lower ν_0 values than PRE-VEB, but not as low as MINWEP.

IV. A CACHE-OBLIVIOUS LOCALITY MEASURE

We have seen how the percentage of block transitions provides a quality measure for a layout given a particular cache block size N. We now remove this dependence on block size and derive a simple measure of locality for graph orderings in a cache-oblivious sense, *i.e.* with no knowledge of cache and line size. Continuing the discussion in Section III-A, we here generalize the measure presented in [14] to weighted graphs.

The observation underlying our cache-oblivious measure is that most block-based caches employed in current computer architectures are hierarchical and nested, with a roughly geometric progression in size. That is, we may write $N = b^k$ for some base b (usually $b = 2$) and positive integer k. We then estimate the total number of cache misses for all k for a particular edge length ℓ as

$$M(\ell) = \sum_{k=1}^{\infty} M_{b^k}(\ell) = \sum_{k=1}^{\lfloor \log_b \ell \rfloor} 1 + \sum_{k=\lfloor \log_b \ell \rfloor + 1}^{\infty} \frac{\ell}{b^k} \\ = \lfloor \log_b \ell \rfloor + \ell \frac{b^{-\lfloor \log_b \ell \rfloor}}{b-1} \tag{4}$$

We note that when ℓ is an exact power of b, $M(\ell)$ simplifies to $\log_b \ell + \frac{1}{b-1}$; otherwise $M(\ell)$ increases monotonically with ℓ. Our primary goal is not to estimate the exact number of cache misses incurred, but rather to assign a relative "cost" as a function of edge length ℓ. We may thus ignore the value of b (since it affects only the slope of M) and the constant term independent of ℓ, and arrive at the approximation

$$M(\ell) \approx \log \ell \tag{5}$$

Intuitively, $\log \ell_{ij}$ measures the number of blocks smaller than ℓ_{ij} that cannot hold both i and j, and thus captures the expected number of block transitions and cache misses associated with ℓ_{ij} in a memory hierarchy.

Finally, if we consider all edges E of the graph, then

$$\begin{aligned} M &= \sum_{k=1}^{\infty} \beta(b^k) = \sum_{k=1}^{\infty} \frac{1}{W} \sum_{ij \in E} w_{ij} M_{b^k}(\ell_{ij}) \\ &= \frac{1}{W} \sum_{ij \in E} w_{ij} \sum_{k=1}^{\infty} M_{b^k}(\ell_{ij}) = \frac{1}{W} \sum_{ij \in E} w_{ij} M(\ell_{ij}) \\ &\approx \frac{1}{W} \sum_{ij \in E} w_{ij} \log \ell_{ij} = \log \nu_0 \end{aligned} \tag{6}$$

gives the *average cache miss ratio*, where ν_0 denotes the *weighted edge product* functional

$$\nu_0 = \exp\left(\frac{1}{W} \sum_{ij \in E} w_{ij} \log \ell_{ij}\right) = \left(\prod_{ij \in E} \ell_{ij}^{w_{ij}}\right)^{1/W} \tag{7}$$

for a weighted graph. In other words, $\nu_0 \approx \exp(M)$. As a result, low values of ν_0 imply good cache utilization across the whole memory hierarchy. As we shall see, this expected behavior is observed also in practice, with layouts optimized for ν_0 having excellent locality properties. (In the unweighted case, $w_{ij} = 1$ and $W = |E|$. We denote the unweighted version of ν_0 by μ_0.) We call the Recursive Layout that minimizes ν_0 (for the weights described in Section III-A) the MINWEP (short for minimum weighted edge product) layout.

A. Other edge-based locality measures

It is important to mention two other locality measures that have been considered in the literature: the average edge length, μ_1, and the maximum edge length, μ_∞. The small example in Figure 7 includes the layouts MINLA [15] in Figure 7m, which minimizes μ_1, and MINBW [16] in Figure 7n, which minimizes μ_∞. Similar to ν_0, which measures the weighted edge length product, we may define the average weighted edge length ν_1. This figure also presents these four statistics (ν_0, ν_1, μ_1, μ_∞) for all layouts discussed in this paper.

Our experiments on block transitions (Figure 5), observed cache misses, and timings indicate that these other layouts (MINLA, MINBW) have significantly worse locality than MINWEP, lending support to our claim that the weighted edge product (represented by ν_0) is the correct measure to consider.

Continuing the discussion in Section III-A, observe that for block sizes larger than the number of elements in the binary tree, $M_N(\ell)$ reduces to $\frac{\ell}{N}$, a linear function of the edge length ℓ. This implies that the probability of a cache miss $\beta(N)$ reduces to $\frac{1}{WN} \sum_{ij \in E} w_{ij} \ell_{ij}$, a weighted sum of the edge lengths. Thus, for very large block sizes, the optimal ordering is one that minimizes ν_1 This is confirmed by Figure 5, where we see that MINWLA, a layout that minimizes ν_1, performs as well as MINWEP for large block sizes.

Based on an empirical study, we conjecture that among all Recursive Layouts ν_1 is minimized by any layout that arranges the outermost top subtree in-order and all other subtrees pre-order, irrespective of the cut height. In our nomenclature, these layouts are denoted by $\mathcal{I}^*_\infty$, where $*$ is a wild-card.

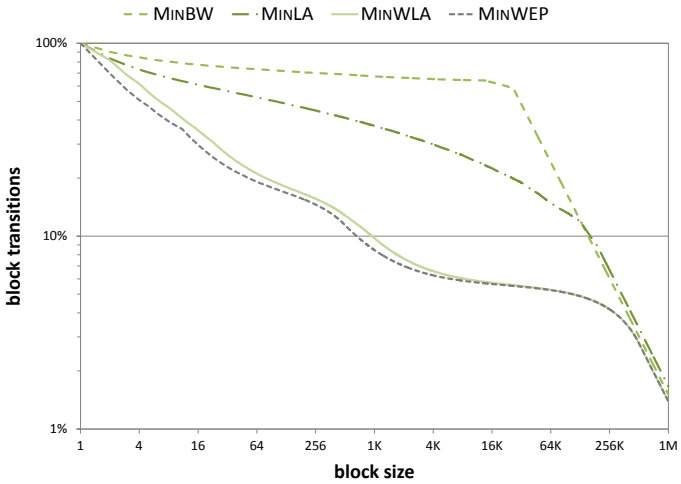


Fig. 5: Block transitions in a tree of height 20 for the layouts that minimize μ_∞ (BW), μ_1 (LA), ν_1 (WLA), ν_0 (WEP).

Consider $\mathcal{I}^1_\infty$, the $\mathcal{I}^*_\infty$ layout with cut height $g = 1$, which we denote as MINWLA (see Table I). Restricting ourselves to Recursive Layouts with cut height $g = 1$, MINWLA provably minimizes ν_1. We delegate all proofs to the extended version [17]. All proofs involving Recursive Layouts with cut height $g = 1$ were proved by induction on the height of the tree.

Theorem 1. *Among all Recursive Layouts with cut height $g = 1$, the* MINWLA *layout minimizes μ_1, ν_1, and the average weighted edge length for all other weight distributions where the weight does not increase as we go down the tree.*

In this paper, we present a succession of Hierarchical Layouts that reduce ν_0, and we see that these also tend to reduce ν_1, μ_1, and μ_∞, suggesting that these might be good layouts in other settings that benefit from better locality. In [18], the authors show that minimizing ν_0 results in compression-friendly layouts. We note that minimizing ν_0 is likely to result in high locality layouts for all graphs, and not just trees. For algorithms designed to minimize μ_0, ν_0, μ_1, and μ_∞, respectively, on general graphs, see [14], [18], [19], and [20].

V. MINIMIZING THE WEIGHTED EDGE PRODUCT

We have shown that layouts with lower Weighted Edge Product ν_0 result in fewer block transitions (measured by β). So far, we have presented IN-VEB, with lower ν_0 values than PRE-VEB. Section V-A shows that ν_0 can be further reduced by *alternating layouts*. Ultimately, the goal is to find the Hierarchical Layout that minimizes ν_0 – the MINWEP layout.

A. Ordering the subtrees: Alternating Hierarchical Layouts

In the discussion so far, we have not yet determined the optimal relative ordering of the bottom subtrees – we have prescribed it to be in the order of the top subtree leaves. A simple way to reduce ν_0 is to reduce the product of edge lengths among all edges that have the same weight, without modifying the lengths of all other edges. If we consider the Hierarchical Layout at a particular branch of the recursion, all the edges between the top subtree and the bottom subtrees have the same weight. By considering such equal-weight edges, the

next result proves that a layout that orders the bottom subtrees in the reverse order of the parent leaves reduces ν_0. In such a layout, the order of the nodes appears to alternate between left-to-right and right-to-left. As a result, we refer to Hierarchical Layouts that arrange the bottom subtrees in the reverse order of the parent leaves as *alternating* Hierarchical Layouts.

Theorem 2. *For any subtree in a particular branch of the recursion, suppose we fix the internal ordering of the leaves of the top subtree A and the arrangement of all the bottom subtrees in subsequent branches of the recursion. Then, the product of all the edge lengths between the top subtree and the bottom subtrees is minimized by ordering the bottom subtrees in reverse order of that of the parent leaves L_A.*

As a corollary of Theorem 2, we see that when the cut height $g > 1$, the optimal relative ordering of bottom subtrees is one that positions both the bottom subtrees of a particular parent leaf in L_A adjacent to each other. This suggests that the initial orderings (PRE-VEB and IN-VEB) got the adjacency of the bottom subtrees right – they only had the order wrong.

By recursive application of Theorem 2, we order the bottom subtrees in the reverse ordering of the parent leaves L_A at each level of recursion, converting any Hierarchical Layout to its alternating version, thus reducing ν_0. We denote the alternating version of IN-VEB by IN-VEBA, and define PRE-VEBA similarly. In our nomenclature, these two layouts are $\widetilde{\mathcal{I}}_1^{\lfloor h/2 \rfloor}$ and $\widetilde{\mathcal{P}}_\infty^{\lfloor h/2 \rfloor}$, respectively (see Table I).

Figure 7c illustrates IN-VEBA for a tree of height 6. Observe that the bottom subtrees are arranged in reverse order of their parent leaves. In the outermost branch of the recursion, the rightmost two leaves in the top subtree are arranged at positions 35 and 33, and the corresponding child subtrees are rooted at positions 39, 46, 53, and 60. That is, the child subtrees are arranged in reverse order of the parent leaves (39 and 46 connected to 35), compared to IN-VEB (see Figure 7e, where 39 and 46 are connected to 33). We see that by alternating, the sum of edge lengths between top and bottom subtrees remains the same, but we have increased their variance, thus reducing their product and consequently ν_0. A similar argument holds for alternating pre-order trees (see Figure 7d and Figure 7f).

It is important to mention that *alternating* a particular layout has no effect on ν_1. However, since the variance of the edge lengths is increased, alternating a layout will increase μ_∞, and may increase the number of unit-length edges. Since ν_0 is a function of the product of the edge lengths, the Weighted Edge Product is reduced. As an example, we can see the effect of alternating a layout on ν_0, ν_1, and μ_∞ by comparing IN-VEBA (Figure 7c) and IN-VEB (Figure 7e).

Figure 4(top left) shows that PRE-VEBA has smaller ν_0 values than PRE-VEB, but this improvement is not as drastic as the improvement from PRE-VEB to IN-VEB. A natural question to ask is: Does an ordering that reduces ν_0 result in better cache-oblivious layouts? And if so, do we get a greater improvement from PRE-VEB to IN-VEB, as predicted by the ν_0 values? We first look to block transition percentages (β) to show that alternating layouts are better. Figure 3(left) plots β for PRE-VEBA and IN-VEBA as a function of block size for a tree of height 20. We see that IN-VEBA is virtually indistinguishable from IN-VEB, whereas PRE-VEBA dominates PRE-VEB for small block sizes. Figure 4(bottom left) also plots β for PRE-VEBA and IN-VEBA as a function of tree height for a variety of block sizes. Again, we see that IN-VEBA is virtually indistinguishable from IN-VEB, but PRE-VEBA dominates PRE-VEB for all tree heights. And we see the same pattern with cache miss rate. Figure 4(top right) also plots the explicit search time for IN-VEBA and PRE-VEBA. Comparing with IN-VEB and PRE-VEB, we see the exact same pattern. IN-VEB and IN-VEBA are indistinguishable from each other, with approximately 5% worse explicit search times than MINWEP. On the other hand, PRE-VEBA is about 5% better than PRE-VEB.

From these experiments, we see that the improvement from PRE-VEB to PRE-VEBA is far less than the improvement from PRE-VEB to IN-VEB. This suggests that while an alternating version always improves the layout (we restrict our attention to alternating layouts for the rest of this paper), it is far more important to switch from pre-order to in-order. One should consider this the main take-home message of this paper: All data structures that use a pre-order Hierarchical Layout should, at the very least, switch to an in-order version of the same Hierarchical Layout. Later, in Section V-C, we will see that this result may depend on the cut height, but not for the cut heights $\lfloor h/2 \rfloor$ that have been used in practice.

B. Constructing hybrid layouts: The HALFWEP layout

We now analyze the impact of varying the position of the top subtree A relative to all the bottom subtrees. Recursive Layouts restrict us to the two extremes represented by PRE-VEBA and IN-VEBA, wherein A is positioned either at one end or in the middle of all the bottom subtrees. However, IN-VEBA and PRE-VEBA arrange **all** bottom subtrees identically, either in-order or pre-order, respectively. We can consider many more permutations by ordering some of the bottom subtrees in-order and others pre-order. As before, the locality measure ν_0 guides us in these decisions. Clearly, IN-VEBA results in smaller ν_0 than PRE-VEBA. Can a hybrid layout (by modifying IN-VEBA, possibly) reduce ν_0 even further?

To construct a hybrid layout, we must take into account the trade-offs involved. First, observe that any bottom subtree is arranged in a contiguous block in memory, and has only one edge connecting it to the rest of the tree – the edge from its root to a leaf in the top subtree. Therefore, rearranging any bottom subtree potentially results in two changes to its contribution to ν_0: the length of the edge connecting its root to its parent, and the lengths of the edges in the subtree itself. Discounting the connection to the top subtree, a bottom subtree ordered as in PRE-VEBA has a larger Weighted Edge Product than when it is ordered as in IN-VEBA. However, the root of a pre-order bottom subtree is closer to its parent than the root of an in-order bottom subtree, and the weight of this edge is larger than the weight of any edge within the bottom subtree. So there are potential benefits to modifying IN-VEBA by arranging some of the bottom subtrees pre-order. This nevertheless raises the question: Which bottom subtrees should we modify, if any? Also, observe that the in-order bottom subtrees are identical, and differ only in their distance to their parent leaf in the top subtree, and similarly for the pre-order trees. As we move further away from the top subtree, the proportional reduction in

the length ℓ of the edge connecting the subtrees decreases (*i.e.* the slope of $\log \ell$ approaches zero), whereas the degradation in the ν_0 value of the bottom subtree remains the same. As a result, the marginal benefit of converting an in-order bottom subtree into a pre-order bottom subtree decreases. Therefore, if arranging any bottom subtree in-order results in lower ν_0 than arranging it pre-order, then this must also be true for all bottom subtrees further away from the top subtree.

To find the best layout, we undertook a detailed empirical study that evaluated all Recursive Layouts for trees up to height 20. We considered all possible cut heights $g \leq \lfloor h/2 \rfloor$ (we quickly determined that larger g were not beneficial). We calculated ν_0 for every layout for each tree height. We noticed that the optimal ordering always arranged the bottom subtrees closest to the top subtree pre-order, arranged all other bottom subtrees in-order, and used an in-order arrangement for the outermost branch of the recursion. In comparison, IN-VEBA arranges all bottom subtrees in-order, and PRE-VEBA arranges all of them pre-order. We give the version of this layout with cut height $g = \lfloor h/2 \rfloor$ the special name HALFWEP. In our nomenclature, HALFWEP is $\widetilde{\mathcal{I}}_2^{\lfloor h/2 \rfloor}$ (see Table I).

Figure 6(top) shows that HALFWEP and MINWEP have almost indistinguishable values of ν_0 and performance in explicit search times, further validating ν_0 as the appropriate locality measure for deriving cache-oblivious layouts.

Figure 7a illustrates HALFWEP for a tree of height 6. Observe that the bottom subtrees closest to the top subtree are arranged pre-order. At the outermost branch of the recursion, the subtrees rooted at positions 28 and 36 are arranged pre-order in HALFWEP. These are arranged in-order in IN-VEBA (see Figure 7c). From the thickness of the edges, one can see that HALFWEP reduces some edge lengths for every branch of the recursion by replacing some in-order bottom subtrees by pre-order bottom subtrees. This does increase some distances within the bottom subtree (the next recursive branch), but deeper down the tree, where they contribute less to the Weighted Edge Product. This is confirmed by the ν_0 values for HALFWEP (1.863) and IN-VEBA (2.322).

Our empirical analysis is also backed by theory, when restricted to certain cut heights. In Theorem 3, we show that when the cuts are made at the top of the tree ($g = 1$) at all branches of the recursion, this HALFWEP-like layout provably minimizes ν_0. We refer to this layout as the MINEP layout, because it also minimizes the edge product μ_0 for unweighted trees. In our nomenclature, MINEP is $\mathcal{I}_2^1$ (see Table I).

Theorem 3. *The* MINEP *layout minimizes* ν_0 *among all Recursive Layouts with cut height* $g = 1$.

C. Optimizing the cut height: The MINWEP layout

In the discussion so far, we have ignored the effect of the cut height by restricting ourselves to the case where $g = \lfloor h/2 \rfloor$. Before we find the optimal cut height, we describe other layouts that turn out to be part of the Hierarchical Layout framework, albeit with extreme cut height values.

Consider cut height $g = 1$. Analogous to how HALFWEP is a hybrid of IN-VEBA and PRE-VEBA, one can think of MINEP as a hybrid of two other simple layouts: the common IN-ORDER and PRE-ORDER depth-first layouts. All three are Hierarchical Layouts that cut every subtree at height $g = 1$, but differ in how the bottom subtrees are arranged. IN-ORDER arranges all subtrees in-order, and PRE-ORDER arranges all subtrees pre-order. In our nomenclature, IN-ORDER is $\mathcal{I}_1^1$ and PRE-ORDER is $\mathcal{P}_\infty^1$ (see Table I). Observe that when the cut height $g = 1$, there is only one leaf node in L_A at every branch of the recursion, and therefore the notion of alternating layouts is not relevant. Furthermore, there are only 2 bottom subtrees at each branch of the recursion, and therefore in-order and pre-order are the only two options for positioning the top subtree. As a result, all Hierarchical Layouts that are described using cut height $g = 1$ at all branches of the recursion are in fact Recursive Layouts. This is not true for other cut heights.

Figure 7g illustrates IN-ORDER for a tree of height 6. Observe that IN-ORDER arranges the two bottom subtrees at the outermost recursion in-order, resulting in their roots being placed at positions 16 and 48. These roots are arranged pre-order at positions 31 and 33 in MINEP (see Figure 7b). On the other hand, PRE-ORDER (see Figure 7h) arranges all subtrees pre-order. The roots of the same bottom subtrees are arranged pre-order at positions 2 and 33. Observe that IN-ORDER and PRE-ORDER have (nearly exactly) the same number of short edge lengths (counting the number of thick lines), but these are at the bottom of the tree for in-order, where the weights are much smaller. This results in much larger ν_0 values for IN-ORDER (4.854), when compared to PRE-ORDER (3.116).

At the other end of the spectrum in terms of cut height is $g = h - 1$, where each subtree is cut one level above the bottom. It turns out that the Hierarchical Layout with $g = h-1$ that arranges all subtrees pre-order (similar to PRE-VEB with cut height $g = \lfloor h/2 \rfloor$) is the simple and commonly used breadth-first order. For this reason, we denote the breadth-first layout as PRE-BREADTH. Figure 7j illustrates the PRE-BREADTH layout for a tree of height 6. Observe that the nodes are arranged by level. Furthermore, one can now also consider in-order and/or alternating variants on the breadth-first ordering. We denote the in-order variant by IN-BREADTH. Observe that when $g = h - 1$, the bottom subtrees are single nodes, and therefore the notion of their arrangement into pre- or in-order is not relevant. In our nomenclature, IN-BREADTH is $\mathcal{I}_*^{h-1}$ and PRE-BREADTH is $\mathcal{P}_*^{h-1}$ (see Table I).

In our detailed empirical analysis, which suggested that ν_0 is minimized by layouts that fit the characterization $\widetilde{\mathcal{I}}_2^*$ in our nomenclature, we noticed that the optimal cut height (denoted by opt) is different from HALFWEP for pre-order subtrees: $g_P^{opt}(h) = \max\{1, \lfloor (h-1)/2 \rfloor\}$, resulting in different cut heights for pre-order subtrees whose height h is an even number greater than 2. For in-order subtrees, it is the same as before, *i.e.*, $g_I^{opt}(h) = \lfloor h/2 \rfloor$. We also noticed one exception to the above rule, with $g_I^{opt}(13) = 7$, and correspondingly $g_P^{opt}(12) = 6$. However, ignoring this exception had a indistinguishable effect on ν_0 or performance, and for this reason we do not include the exception in the definition of MINWEP.

Based on these experiments, we define MINWEP as the HALFWEP-like layout with the cut heights presented above. In our nomenclature, MINWEP is $\widetilde{\mathcal{I}}_2^{opt}$ (see Table I). For trees of height $h \leq 6$, MINWEP is identical to HALFWEP (see Figure 7a). This is because such trees have no pre-order subtrees of heights that are even numbers greater than 2.

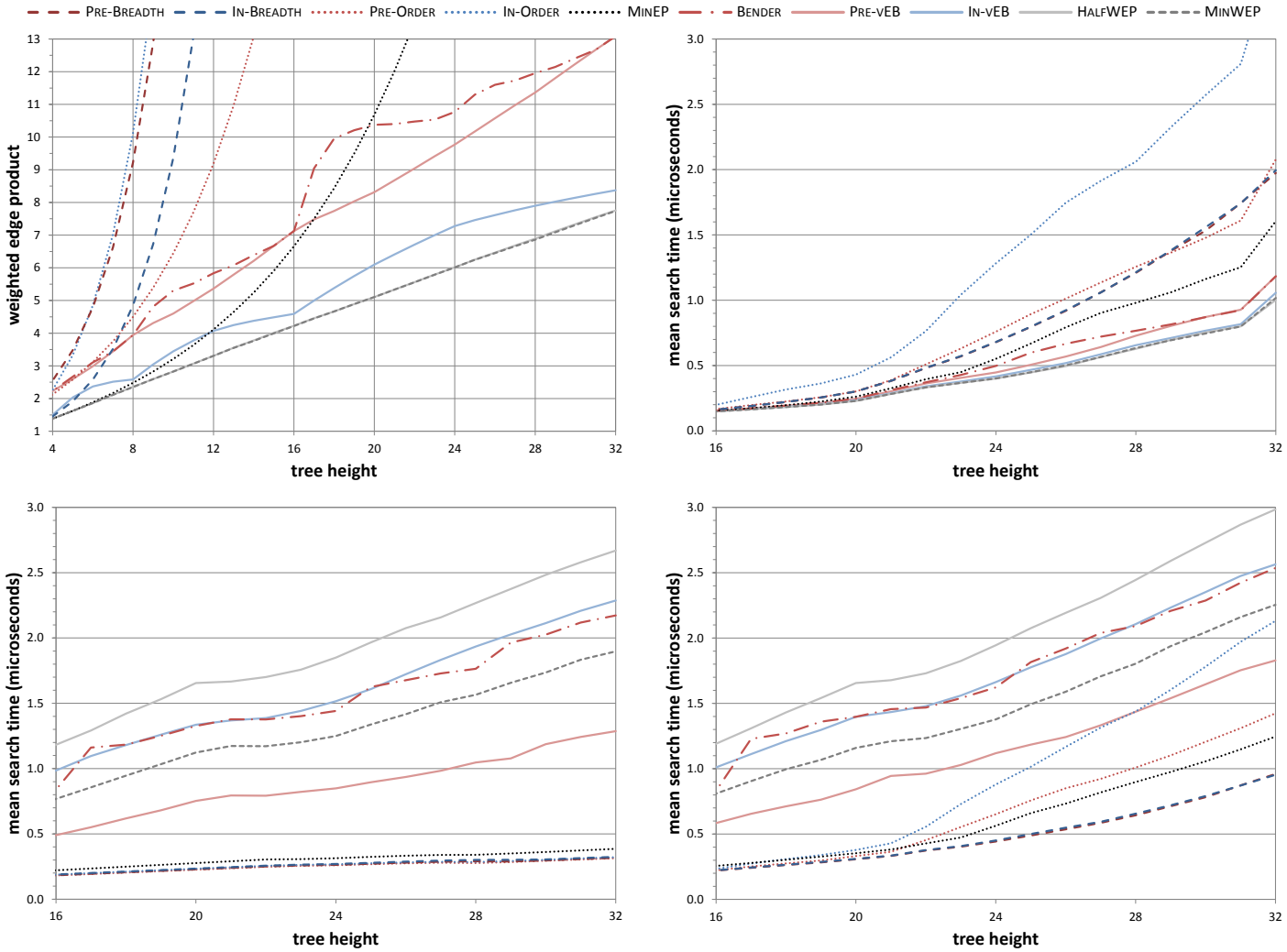


Fig. 6: Performance metrics as a function of tree height for several layouts. Top left: Weighted edge length product ν_0. Top right: Pointer-based search time. Bottom left: Pointer-less search time excluding all memory accesses. Bottom right: Pointer-less search time.

Observe that the cut height for an in-order subtree can be calculated directly from the pre-order cut height as follows: $g_I^{opt}(h) = 1$ if $h = 2$, and $g_P^{opt}(h-1)+1$ otherwise. Analyzing Hierarchical Layouts where the closest bottom subtree is arranged pre-order and the cut heights are chosen such that $g_I(h) = g_P(h-1)+1$, we see that cutting all in-order subtrees at height $g_I(h) = 1$ instead results in the same layout. This is because this in-order cut results in two pre-order bottom subtrees of height $h-1$, each of which will subsequently be cut at the same height as they would have been when part of an in-order subtree of height h. Since the closest bottom subtree is pre-order in either case, the layouts are identical. As a result, we can set $g_I^{opt}(h) = 1$. As we shall see later, this is important since it simplifies the index computation for pointer-less trees. Note that this optimization cannot be applied to HALFWEP.

D. The cost of cache misses: Explicit pointer-based searches

We observed earlier that HALFWEP and MINWEP are virtually indistinguishable in terms of explicit search time. This is because they are exactly the same ordering schemes, but with very slightly different cut heights. Larger differences in cut heights can make a significant difference. Consider the values of ν_0 in Figure 6(top left) for many of the layouts presented so far. Recall that BENDER and PRE-VEB differ from each other only in the choice of the cut height g. For BENDER, the cut height $g = h - 2^{\lceil \log_2(h/2) \rceil}$, which is identical to PRE-VEB ($g = \lfloor h/2 \rfloor$) only for subtree heights that are a power of two. As expected, we see identical values of ν_0 for BENDER and PRE-VEB for trees of height 4, 8, 16, and 32. However, for all other tree heights, BENDER gives higher values for ν_0; sometimes 20% worse. This manifests itself in similarly worse pointer-based search times compared to PRE-VEB. This suggests that for a particular ordering scheme, the optimal cut height is closer to halfway down the tree.

Cut heights $g = 1$ and $g = h - 1$ illustrate this further. MINEP, which is identical to MINWEP except in its choice of the cut height ($g = 1$), results in significantly different trees (especially for larger tree heights), and we observe a steep divergence in ν_0 as the tree height increases. As expected, MINEP's performance (measured using pointer-based search times) also degrades significantly for large tree heights. At the other end of the spectrum, consider PRE-BREADTH and

IN-BREADTH, which are identical to PRE-VEB and IN-VEB, respectively, except in the choice of cut height. A cut height of $g = h - 1$ results in significantly different layouts, especially for large tree heights. Even for the small example in Figure 7, we see that PRE-BREADTH is quite different from PRE-VEB. In Figure 6(top right), we see that the pointer-based search time is significantly worse for breadth-first layouts, when compared to PRE-VEB and IN-VEB.

From the ν_0 values in Figure 6(top left), we also observe that in-order is not always better than pre-order. For a cut height of $g = 1$, PRE-ORDER results in much smaller ν_0 values than IN-ORDER. The example in Figure 7 suggests why: All the short edges in IN-ORDER are near the bottom of the tree, where the contribution to ν_0 is minimal. However, this behavior changes as we increase the cut height, and at some point, in-order layouts are better than pre-order layouts. When the cut is approximately near halfway down the tree, in-order layouts such as IN-VEB result in much smaller ν_0 values than pre-order layouts such as PRE-VEB. As we increase the cut height all the way to $g = h - 1$, we observe that the in-order version of the breadth-first layout IN-BREADTH continues to be better than the pre-order version PRE-BREADTH.

E. The computational cost of layouts: Pointer-less searches

Based on explicit pointer-based search times, we have shown that MINWEP is a cache-oblivious layout with almost 20% improvement in performance when compared to the best in the literature, represented by PRE-VEB. However, MINWEP is a more complex layout than PRE-VEB. The natural question therefore is: If we considered implicit, pointer-less search times, would MINWEP still compare favorably with PRE-VEB? In [5], the authors showed that for small tree heights, even layouts that have poor cache-performance such as IN-ORDER and PRE-BREADTH perform better than PRE-VEB in implicit search, simply because it is trivial to compute the position of a node in such layouts.

To understand the trade-offs involved, we first measured the time taken to compute the index of child nodes in a pointer-less search by excluding all memory accesses.[1] Listing 1 lists the code that takes the PRE-BREADTH index for a node and computes its corresponding MINWEP index. This code needs to be executed for every transition in the search tree. Here the depth (level) $d = \lfloor \log_2 i \rfloor$ of the node is maintained together with i along the search path from the root.

In Figure 6(bottom left), we see that the index computation time is almost constant for simple layouts (IN-ORDER, PRE-ORDER, IN-BREADTH, and PRE-BREADTH). The slow increase merely stems from the longer search paths as the height of the tree is increased. Furthermore, MINWEP's index computation time is usually 4 times that of the simple layouts. Comparing MINWEP with the van Emde Boas layouts (IN-VEB, PRE-VEB, BENDER, HALFWEP) is more interesting. Not surprisingly, HALFWEP performs worse than IN-VEB on this metric (around 20% worse), since it is a more complex layout. Observe that PRE-VEB performs better than IN-VEB by almost 50%. It turns out that the index can be computed more quickly within a pre-order subtree, since one does not need to keep track of left and right, and also because some other optimizations unique to pre-order layouts can be performed. This observation is key, since it allows us to compute the index for MINWEP in 30% less time than HALFWEP. This is because we can set $g_I^{opt}(h) = 1$, as shown in Section V-C, reducing the computational burden significantly by converting any in-order computation to a pre-order computation. As a result of this optimization, mean index computation times for MINWEP are also about 20% less than those of IN-VEB. Finally, observe that index computations take 60% more time in BENDER compared to PRE-VEB, because of the additional time spent computing BENDER's complex cut heights.

Figure 6(bottom right) also presents our results on implicit, pointer-less search times. One can think of these as a combination of the index computation times (which do not include memory accesses) and explicit search times (which include memory accesses, but avoid index computations using pointers). We see that for the more complex layouts, the implicit search times correlate very well with the index computation times. This is because of the relatively fast memory access times; if we added disk or even flash to the memory hierarchy, we would expect the relative order among the implicit times to be similar to the explicit times. The only perceptible difference in our experiments is that the pre-order layouts (PRE-VEB and BENDER) perform slightly worse, since they perform almost 20% worse on the explicit search times. Among the simpler layouts, the implicit search times diverge significantly from the index computation times due to their poor memory access patterns. For tress of height 28, IN-ORDER already performs worse than PRE-VEB, and we expect all of the simpler layouts to perform worse than MINWEP as the height of the tree increases beyond 32.

F. Experimental Setup

Our experiments were run on a single core of a dual-socket 6-core 2.80 GHz Intel Xeon X5660 (Westmere-EP) processor with 96 GB of 3x DDR3-1333 RAM split over two 48 GB NUMA memory banks, 12 MB 16-way per-socket shared L3 cache, 256 KB 8-way L2 cache, and 32 KB 8-way L1 data cache. All three caches use 64-byte cache lines. To reduce noise in the timing measurements, we computed the median time of 15 runs. Each run searches for (up to) 10 million nodes, selected randomly. We counted the number of L1 and L2 cache misses incurred in memory accesses to the binary tree (stored as a linear array) using `valgrind-3.5.0`. We also repeated our experiments on different architectures, from powerful workstations to laptops, and observed similar results.

VI. CONCLUSIONS

In this paper, we present MINWEP, a new layout for cache-oblivious search trees that outperforms layouts used in the literature by almost 20%. Using a general framework of Hierarchical Layouts, we show that MINWEP minimizes a new locality measure ν_0 (representing the Weighted Edge Product) that correlates very well with cache misses in a multi-level cache hierarchy. All widely used cache-oblivious versions of search trees rely on van Emde Boas layouts, which are shown to be a special case of Hierarchical Layouts. Therefore, the performance of all these data structures can be easily improved by switching to layouts derived from MINWEP.

[1] We achieved this by storing the keys $\{1, \ldots, |V|\}$ in the tree, allowing them to be easily inferred without lookup via their in-order index.

	Cut height g	Pre-order layouts	Hybrid layouts		In-order layouts
		$\mathcal{P}_\infty$	$\mathcal{I}_\infty$	$\mathcal{I}_2$	$\mathcal{I}_1$
Depth-first	1	PRE-ORDER ($\mathcal{P}_\infty^1$)	MINWLA ($\mathcal{I}_\infty^1$)	MINEP ($\mathcal{I}_2^1$)	IN-ORDER ($\mathcal{I}_1^1$)
Other		BENDER ($\mathcal{P}_\infty^{h-2^{\lceil \log_2(h/2) \rceil}}$)		MINWEP ($\widetilde{\mathcal{I}}_2^{opt}$)	
van Emde Boas	$\lfloor h/2 \rfloor$	PRE-VEB ($\mathcal{P}_\infty^{\lfloor h/2 \rfloor}$)			IN-VEB ($\mathcal{I}_1^{\lfloor h/2 \rfloor}$)
		PRE-VEBA ($\widetilde{\mathcal{P}}_\infty^{\lfloor h/2 \rfloor}$)		HALFWEP ($\widetilde{\mathcal{I}}_2^{\lfloor h/2 \rfloor}$)	IN-VEBA ($\widetilde{\mathcal{I}}_1^{\lfloor h/2 \rfloor}$)
Breadth-first	$h-1$	PRE-BREADTH ($\mathcal{P}_*^{h-1}$)			IN-BREADTH ($\mathcal{I}_*^{h-1}$)

TABLE I: Nomenclature for Hierarchical Layouts. The table summarizes the layouts discussed in the text, organized by cut height (rows) and subtree ordering (columns). The cut height function g^{opt} for MINWEP is described in Section V-C. The wild-card $*$ indicates that a particular parameter is not relevant.

```
uint index(uint i, uint d, uint h) {      // BF index i, node depth d, tree height h
    uint p = 1 << --h;                    // MinWEP index being computed
    while (d) {                           // iterate until node is root of subtree
        uint q = (i >> --d) & 1;          // initial offset (pre: q=1; post: q=0)
        uint r = q - 1;                   // bit reversal (pre: r=0; post: r=~0)
        i ^= r;                           // post-order is reversal of pre-order
        while (d) {                       // iterate until node is root of subtree
            uint g = max(1, (h - 1) / 2); // top subtree height
            if (d < g) {                  // is node in top subtree?
                h = g;                    // set height to top subtree height
                i = ~i;                   // alternate left/right ordering
            } else {                      // node is in bottom subtree
                h -= g;                   // bottom subtree height
                d -= g;                   // depth within bottom subtree
                uint m = (1 << g) - 1;    // number of nodes in top subtree
                q += m;                   // advance past top subtree
                uint k = (i >> d) & m;    // subtree number (pre: k=0; in: 1<=k<=m)
                if (k) {                  // in in-order subtree?
                    q += (k << h) - k;    // advance past k bottom subtrees
                    q += (1 << --h) - 1;  // advance to root of in-order subtree
                    break;                // transition to in-order case
        }   }   }
        i ^= r;                           // restore i if post-order
        q ^= r;                           // negate offset if post-order
        p += q;                           // advance to smaller in-order subtree
    }
    return p;                             // return MinWEP index
}
```

Listing 1: Breadth-first to MINWEP index translation.

While enumerating all possible orderings for small trees, we noticed that the optimal ν_0 value is sometimes obtained by layouts that do not place the top subtree at one end or in the middle of the bottom subtrees. This implies that Recursive Layouts do not necessarily optimize ν_0. One direction of future study is to generalize the notion of Recursive Layouts to include such Hierarchical Layouts, and to construct unrestricted layouts that optimize ν_0. We would also like to prove that, at least among all Recursive Layouts, MINEP and MINWEP minimize μ_0 and ν_0, respectively, since we believe this is true based on our extensive empirical study.

REFERENCES

[1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[2] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal prefetching B$^+$-trees: Optimizing both cache and disk performance," in *ACM SIGMOD International Conference on Management of Data*, 2002, pp. 157–168.

[3] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "Cache-oblivious string B-trees," in *ACM Symposium on Principles of Database Systems*, 2006, pp. 233–242.

[4] H. Prokop, "Cache-oblivious algorithms," Master's thesis, Massachusetts Institute of Technology, 1999.

[5] G. S. Brodal, R. Fagerberg, and R. Jacob, "Cache oblivious search trees via binary trees of small height," in *ACM-SIAM Symposium on Discrete Algorithms*, 2002, pp. 39–48.

[6] M. A. Bender, E. D. Demaine, and M. Farach-Colton, "Cache-oblivious B-trees," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 341–358, 2005.

[7] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz, "The cost of cache-oblivious searching," *Algorithmica*, vol. 61, no. 2, pp. 463–505, 2011.

[8] M. A. Bender, H. Hu, and B. C. Kuszmaul, "Performance guarantees for B-trees with different-sized atomic keys," in *ACM Symposium on Principles of Database Systems*, 2010, pp. 305–316.

[9] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming B-trees," in *ACM Symposium on Parallel Algorithms and Architectures*, 2007, pp. 81–92.

[10] R. Pagh, Z. Wei, K. Yi, and Q. Zhang, "Cache-oblivious hashing," in *ACM Symposium on Principles of Database Systems*, 2010, pp. 297–304.

[11] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, "Cache-oblivious mesh layouts," *ACM Transactions on Graphics*, vol. 24, no. 3, pp. 886–893, 2005.

[12] M. A. Bender, B. C. Kuszmaul, S.-H. Teng, and K. Wang, "Optimal cache-oblivious mesh layouts," *Theory of Computing Systems*, vol. 48, no. 2, pp. 269–296, 2011.

[13] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't thrash: How to cache your hash on flash," *PVLDB*, vol. 5, no. 11, pp. 1627–1637, 2012.

[14] S.-E. Yoon and P. Lindstrom, "Mesh layouts for block-based caches," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1213–1220, 2006.

[15] F. R. K. Chung, "A conjectured minimum valuation tree," *SIAM Review*, vol. 20, no. 3, pp. 601–603, 1978.

[16] R. Heckmann, R. Klasing, B. Monien, and W. Unger, "Optimal embedding of complete binary trees into lines and grids," in *Graph-Theoretic Concepts in Computer Science*, ser. Lecture Notes in Computer Science, 1992, vol. 570, pp. 25–35.

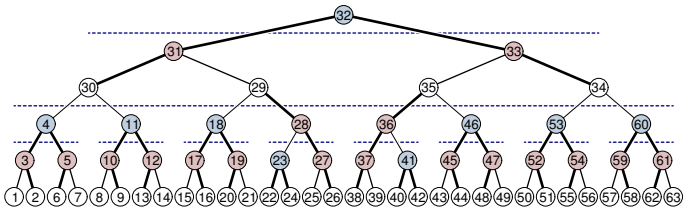
[17] P. Lindstrom and D. Rajan, "Optimal hierarchical layouts for cache-oblivious search trees," Tech. Rep. LLNL-CONF-641294, 2013. [Online]. Available: http://arxiv.org/abs/1307.5899

[18] I. Safro and B. Temkin, "Multiscale approach for the network compression-friendly ordering," *Journal of Discrete Algorithms*, vol. 9, no. 2, pp. 190–202, 2011.

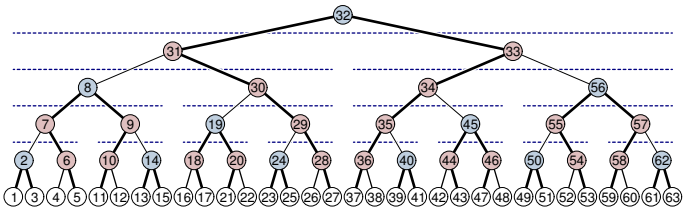
[19] I. Safro, D. Ron, and A. Brandt, "Multilevel algorithms for linear ordering problems," *Journal of Experimental Algorithmics*, vol. 13, pp. 4:1.4–4:1.20, 2009.

[20] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *24th National Conference*, 1969, pp. 157–172.

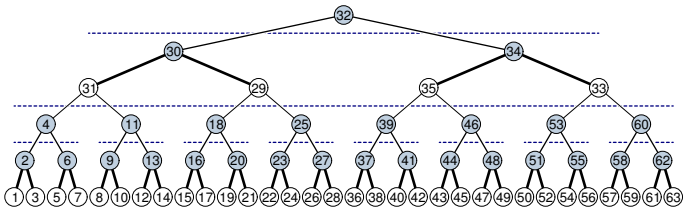
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.



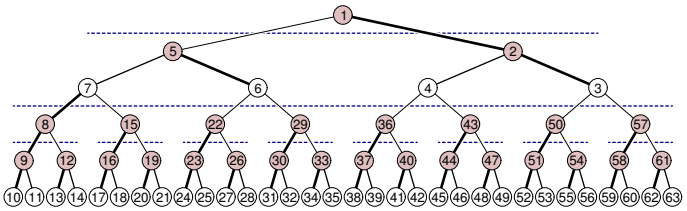
(a) MINWEP = HALFWEP: $\nu_0 = 1.863, \nu_1 = 4.140, \mu_1 = 3.097, \mu_\infty = 26$



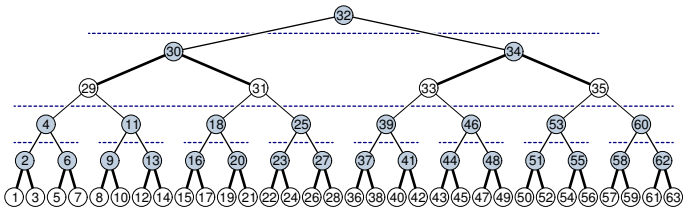
(b) MINEP: $\nu_0 = 1.885, \nu_1 = 4.419, \mu_1 = 2.581, \mu_\infty = 23$



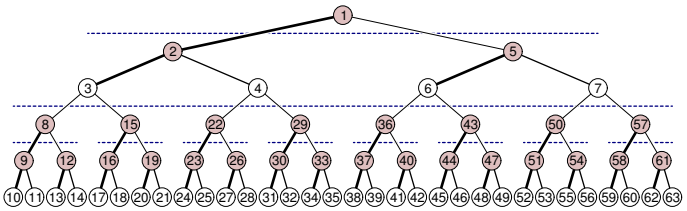
(c) IN-VEBA: $\nu_0 = 2.322, \nu_1 = 4.574, \mu_1 = 3.161, \mu_\infty = 27$



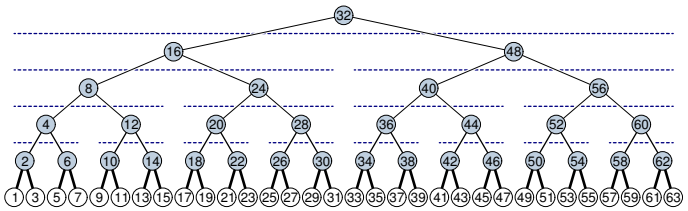
(d) PRE-VEBA: $\nu_0 = 2.835, \nu_1 = 7.100, \mu_1 = 5.145, \mu_\infty = 54$



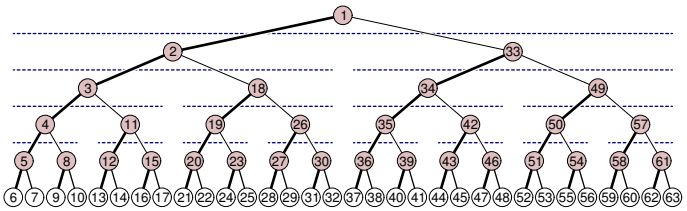
(e) IN-VEB: $\nu_0 = 2.371, \nu_1 = 4.574, \mu_1 = 3.161, \mu_\infty = 25$



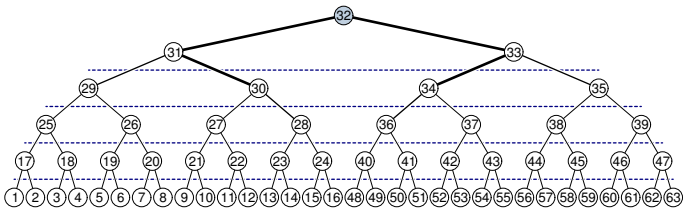
(f) PRE-VEB: $\nu_0 = 2.987, \nu_1 = 7.570, \mu_1 = 5.145, \mu_\infty = 50$



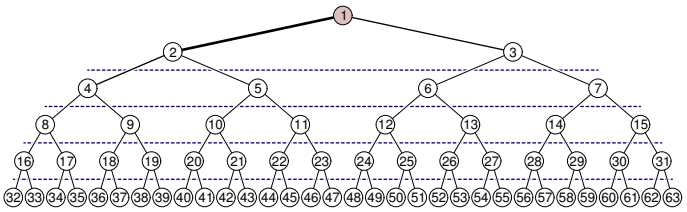
(g) IN-ORDER: $\nu_0 = 4.854, \nu_1 = 7.070, \mu_1 = 2.581, \mu_\infty = 16$



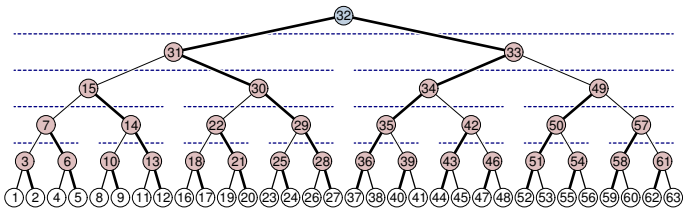
(h) PRE-ORDER: $\nu_0 = 3.116, \nu_1 = 7.570, \mu_1 = 3.081, \mu_\infty = 32$



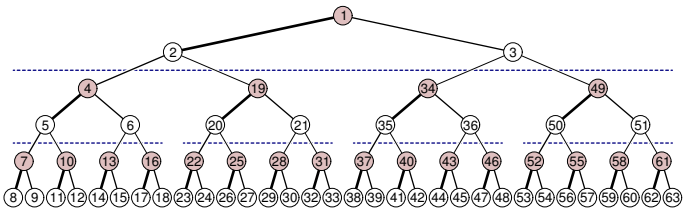
(i) IN-BREADTH: $\nu_0 = 2.580, \nu_1 = 3.845, \mu_1 = 8.258, \mu_\infty = 16$



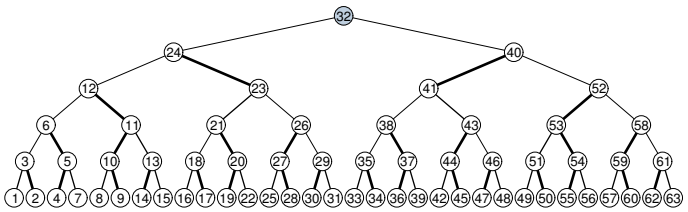
(j) PRE-BREADTH: $\nu_0 = 4.790, \nu_1 = 10.620, \mu_1 = 16.500, \mu_\infty = 32$



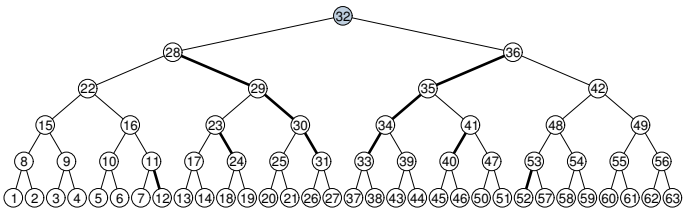
(k) MINWLA: $\nu_0 = 2.054, \nu_1 = 3.845, \mu_1 = 2.581, \mu_\infty = 16$



(l) BENDER: $\nu_0 = 3.106, \nu_1 = 7.570, \mu_1 = 4.113, \mu_\infty = 46$



(m) MINLA: $\nu_0 = 3.061, \nu_1 = 4.620, \mu_1 = 2.323, \mu_\infty = 12$



(n) MINBW: $\nu_0 = 3.554, \nu_1 = 4.295, \mu_1 = 4.581, \mu_\infty = 7$

Fig. 7: Layouts and locality functionals ν_0 (weighted edge product), ν_1 (weighted edge sum), μ_1 (mean edge length), and μ_∞ (maximum edge length) of a tree with $h = 6$ levels. For $h \leq 6$, MINWEP and HALFWEP coincide. Edges ij are drawn with thickness inversely proportional to length ℓ_{ij}. Cuts are shown as dashed lines that span the width of subtrees with 3 or more levels. Colored vertices are roots of in- (blue) or pre-order (red) subtrees with 2 or more levels.